

# nomodo - Manage your system by yourself

Autori (in ordine alfabetico): Giuseppe Glorioso Lucia Polizzi

April 29, 2018

# Contents

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Introduzione al progetto . . . . .	2
1.2	Informazioni tecniche . . . . .	2
<b>2</b>	<b>Backend</b>	<b>2</b>
2.1	utilities . . . . .	4
2.1.1	mongolog() . . . . .	4
2.1.2	changemongologstatus() e funzioni collegate . . . . .	5
2.1.3	mongologstatuserr . . . . .	6
2.1.4	command_success . . . . .	6
2.1.5	command_error . . . . .	6
2.1.6	filedit . . . . .	7
2.1.7	filediff . . . . .	7
2.2	Utenti . . . . .	7
2.3	Network . . . . .	7
2.4	Cron . . . . .	7
2.5	Sistema . . . . .	7
2.6	apache . . . . .	7
2.7	Database . . . . .	7
2.8	File . . . . .	7
2.9	Logs . . . . .	7
<b>3</b>	<b>Frontend</b>	<b>7</b>
<b>4</b>	<b>Utility</b>	<b>7</b>
4.1	Red Hat Developer Toolset . . . . .	7
4.2	kerberos5 . . . . .	7
4.3	Rimossi tra CentOS 6 e 7 e le cui alternative non presenti su CRESCO 6 . . . . .	8
4.4	Rimossi da Centos 6 e 7 e le cui alternative sono presenti su CRESCO 6 . . . . .	10

# 1 Introduzione

## 1.1 Introduzione al progetto

Il progetto nomodo nasce dalla necessità di un applicativo di gestione dei sistemi Ubuntu che sia più immediato ed accessibile rispetto al classico terminale, e quindi utilizzabile anche dagli utenti che per un motivo o per un altro non possono o non vogliono avere a che fare con il terminale. Nomodo si prende in carico di eseguire tutte le chiamate al terminale o meno per eseguire operazioni atte alla gestione del sistema presentando all'utente una interfaccia web chiara e comprensibile. Per operazioni in questo caso si intendono l'aggiornamento, la manutenzione e il miglioramento del sistema come ad esempio l'installazione dei pacchetti, la ricerca e la modifica dei file, così come operazioni di più alto livello come la gestione basilare del web server Apache.

## 1.2 Informazioni tecniche

**Python + Flask** L'applicativo scritto in python è basato sul framework Flask, utilizzato tra l'altro come webserver per l'accesso al pannello. Durante la fase di sviluppo si è utilizzato nginx come reverse proxy in modo da poter raggiungere il pannello web sulla porta 80 e non sulla 5000. È stata presa poi in seguito la decisione di lasciare che l'applicativo girasse sulla porta 5000 in quanto meno comune e quindi meno alla mercé degli hacker.

L'applicazione è stata quindi divisa in modo netto nelle due componenti fondamentali, il Frontend e il Backend che anche andremo quindi ad analizzare qui brevemente e più approfonditamente nei capitoli successivi:

- Il **backend** consiste in una serie di funzioni raccolte in una serie di file a mò di libreria, risiedenti nella cartella `systemcalls` (come ad es. `system.py` o `user.py`), utilizzati sia per la raccolta di dati sia per eseguire azioni sul sistema che non necessitano di output in uscita
- Il **frontend** rappresenta la parte grafica dell'applicativo web, e utilizza le funzioni del backend per la ricerca di informazioni e per la modifica alle componenti del sistema inclusa la modifica dei file quali i file di configurazioni

**MongoDB** Ogni operazione sensibile effettuata tramite l'applicazione comporta la memorizzazione delle modifiche che comporta la stessa in documento di mongodb, così da poter risalire alla storia delle operazioni effettuate e tentare un revert delle modifiche in caso ad esempio il sistema perda di stabilità o le modifiche non portino al risultato sperato. Tali operazioni possono riguardare ad esempio la modifica di un file, o la rimozione di un pacchetto dal sistema. Ogni log in mongodb presenta inoltre un flag **status** che indica se l'operazione eseguita sia andata a buon fine o meno, in modo da rendere più chiara la navigazione tra i log e dare la possibilità all'utente di filtrarli in base a questo campo. <sup>1</sup>

# 2 Backend

Come anticipato in sezione 1.2 il backend è composto da una serie di funzioni raggruppate per categoria che fanno utilizzo di varie librerie python per compiere operazioni che possono o meno alterare lo stato del sistema. Allo stato attuale le categorie che compongono il backend sono le seguenti:

---

<sup>1</sup> Le uniche operazioni memorizzate nel database sono quelle relative all'utilizzo dell'applicativo; una modifica effettuata direttamente sul sistema ad esempio tramite il terminale va incontro alle regole del sistema Ubuntu e ogni modifica potrebbe essere irreversibile. In questi casi fare riferimento ai log del sistema che è possibile trovare al percorso `/var/log/` o sul pannello web alla sezione Log.

- Utenti
- Network
- Cron
- Sistema
- Apache
- Database
- File
- Logs

**Interfaccia al frontend** Ogni funzione chiamata restituisce sempre un dizionario contenente almeno n codice di ritorno e il logid del documento inserito in mongo con un mongo `_id` se applicabile<sup>2</sup> oppure un logid `None` se non è stato creato alcun log. Distinguiamo quindi 2 casi in base al valore della variabile `returncode`:

- Se `returncode = 0` il dizionario conterrà una terza variabile `data` che conterrà i dati richiesti se la funzione chiamata è tesa per restituire output
- Se `returncode  $\neq$  0`

i dati che si sono richiesti se il codice di ritorno è 0 o una variabile `stderr` contenente un messaggio di errore se il codice di ritorno è diverso da 0. Il frontend o l'utente che voglia chiamare per qualsivoglia motivo le funzioni del backend direttamente, potrà farlo quindi nel seguente modo::

```
data = getifacestat()
if data['returncode'] is 0:
    data = data['data']
else:
    print( data['stderr'] )

pprint(data)
```

**subprocess** Le funzionalità di Python più utilizzate per la realizzazione del progetto sono senza dubbio quelle appartenenti alla libreria `subprocess`, che permette di eseguire comandi come se si stessero eseguendo in `bash`. Si è cercato il più possibile di limitare l'utilizzo di questa libreria ma le sue funzionalità si sono rese necessarie nella maggior parte dei casi delle funzioni del backend, a causa della scarsa agilità che ha python di interfacciarsi col sistema sottostante. In generale l'esecuzione di un comando avviene nel seguente modo:

```
command = [ 'ifconfig', '-a' ]
try:
    output = check_output(command, stderr=PIPE, universal_newlines=
        True)
except CalledProcessError as e:
```

---

<sup>2</sup> Cioè in caso l'operazione sia una operazione sensibile e richieda quindi un inserimento in mongo per tenere traccia della stessa

```

        return command_error(e, command, logid)

return command_success( output )

```

Tutte le funzioni di nomodo ritornano o con un `command_success` in caso l'operazione sia andata a buon fine o con un `??` in caso il comando non vada a buon fine e venga lanciata l'eccezione `CalledProcessError`. In entrambi i casi viene restituito un dizionario contenente

- Nel secondo caso sce quindi un dizionario contenente un `returncode` diverso da 0 (quindi errore)

e un messaggio di errore che si trova nella chiave `stderr` oltre al logid dell'operazione se applicabile e al comando lanciato per causare l'errore.

In questo specifico caso invece se l'operazione va buon fine viene invocata la funzione `command_success` che restituisce un `returncode` uguale a 0, e l'output sulla shell nella chiave di dizionario `data` e `logid` se applicabile.

In caso invece il comando che si va ad eseguire non restituisce output la chiave `data` sarà nulla mentre se l'operazione è una operazione sensibile sarà restituito il `logid` di mongo. Un esempio di quest'ultimo caso è rappresentato dalla funzione `removeuser`:

```

def removeuser( user , removehome=None ):

    logid = mongolog( locals() , getuser( user ) )

    try:
        command = [ 'deluser ' , user ]
        if removehome: command.append( '--remove-home' )

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error(e, command, logid)

    return command_success(logid)

```

Nelle prossime sezioni verranno analizzate tutte le categorie e spiegato il funzionamento di ogni funzione che contengono.

## 2.1 utilities

Questa categoria contiene per la maggior parte funzioni che non vengono mai richiamate direttamente dal frontend, ma vengono utilizzate dalle altre funzioni del backend. Fa eccezione invece la funzione `filedit` utile alla modifica di file.

Analizziamo adesso le funzioni di questa libreria.

### 2.1.1 mongolog()

```

def mongolog(params , *args ):

    dblog = dict( {
        'date': datetime.datetime.utcnow() ,      #Operation date

```

```

        'funname': inspect.stack()[1][3],      #Function name
        'parameters': params,                  #Called function's
            parameters
    })

    for arg in args:
        dblog.update( arg )

#ObjectID in mongodb
    return db.log.insert_one( dblog ).inserted_id

```

Viene chiamata da ogni funzione che va a modificare lievemente o pesantemente il sistema e fa un insert su mongodb per memorizzare le informazioni. Accetta N parametri di cui **il primo** (che deve essere sempre presente) è la lista di parametri con cui è stata lanciata la funzione che ha chiamato mongolog. Ad esempio in

```

def ifacedown( iface ):
    logid = mongolog( locals() )
    ...

```

a mongolog viene passata locals() che contiene il parametro iface e che andrà quindi ad inserire nel dizionario di base che restituirà, mentre **il secondo** parametro è opzionale e può essere uno o più dizionari da unire a questo dizionario restituito.

Il dizionario di base è formato da tre elementi:

- La data in cui viene effettuata l'operazione
- Il nome della funzione che ha chiamato mongolog, ricavata tramite il supporto della libreria inspect
- I parametri della funzione che chiama, come spiegato in precedenza, e ottenuti chiamando la funzione locals()

### 2.1.2 changemongologstatus() e funzioni collegate

```

def changemongologstatus(logid):

    return db.log.update_one(
        { '_id': logid },
        { '$set': { 'status' : 'error' } },
        upsert=False
    )

def mongologstatuserr(logid, status='error'):
    return changemongologstatus(logid, status)
def mongologstatussuc(logid, status='success')

    return changemongologstatus(logid, status) #riprenderedaqui

```

Questa funzione è intesa per modificare il campo **status** di un documento di MongoDB. Le funzioni di nomodo (come è giusto che sia) creano un documento di mongo per memorizzare le informazioni sull'operazione prima di procedere all'operazione stessa. In caso un'operazione non andasse nel modo aspettato quindi bisognerebbe marcare il documento appena creato in mongo in modo da avvisare l'utente che sta consultando il log che l'operazione riferita a quel documento non è andata a buon fine; è proprio qui che interviene il campo **status**. Alla creazione del documento questo campo è **success** per poi essere modificato a seconda di come è andata l'operazione

### 2.1.3 mongologstatuserr

### 2.1.4 command\_success

```
def command_success( data=None, returncode=0 ):
    return dict({
        'returncode': returncode ,
        'data': data
    })
```

La funzione **command\_success** viene chiamata ogni qualvolta una funzione del backend inclusa nelle altre categorie esegue l'operazione richiesto in modo corretto ed è quindi in grado di informare l'utente del successo dell'operazione. Questa funzione (così come **command\_error**) viene chiamata da quasi tutte le funzioni al return; fanno eccezione le funzioni che per un motivo o per un altro non possono utilizzarla. In tal caso queste andranno a generare un dizionario proprio da restituire al frontend che avrà però la stessa struttura del dizionario restituito da questa funzione.

**Parametri** Accetta due parametri:

- Il primo che è **data** contiene i dati effettivi da restituire all'utente. Come già detto se l'operazione genera output allora conterrà questo output, altrimenti il logid di mongodb e infine se proprio non c'è niente da restituire verrà una variabile vuota
- Il secondo che non è obbligatorio, è il **returncode**. Di base questa funzione dovrebbe restituire sempre 0 ma è stata data la possibilità di inserire returncode diversi da 0 per implementazioni (improbabili) future.

**Costruisce il dizionario da restituire**

**Return** Restituisce un dizionario contenente un **returncode** e una variabile **data** che assume il tipo dei dati da restituire, da utilizzare come spiegato in sezione 2.

### 2.1.5 command\_error

```
def command_error(e, c):

    return dict({
        'returncode': e.returncode ,
        'command': ' '.join(c) ,
        'stderr': e.stderr
```

```
} )
```

`command_error` è l'opposto di `command_success`. Come visto in sezione 2 viene invocato quando un comando lanciato attraverso la libreria `subprocess` fallisce nell'esecuzione.

**Parametri** Accetta 2 parametri di cui:

- Il primo è l'oggetto creato al lancio dell'eccezione `CalledProcessError`
- Il secondo è il comando che è andato in errore

**Funzionamento** Costruisce il dizionario da restituire inserendovi il `returncode` dell'eccezione lanciata, il comando che ha causato l'eccezione unito in una sola stringa usando il comando `join` e il messaggio di errore così come restituito dall'eccezione

**Return** Restituisce il dizionario creato come descritto nel funzionamento.

#### 2.1.6 fledit

#### 2.1.7 filediff

### 2.2 Utenti

### 2.3 Network

### 2.4 Cron

### 2.5 Sistema

### 2.6 apache

### 2.7 Database

### 2.8 File

### 2.9 Logs

## 3 Frontend

Frontend

## 4 Utility

### 4.1 Red Hat Developer Toolset

### 4.2 kerberos5

1. Copiare il file `/etc/krb5.conf` da uno dei nodi di cresco 4
2. Abilitare kerberos come tipologia di autenticazione:  
`authconfig --enablekrb5 --updateall`
3. Utilizzare `klog.krb5` come `klog` di default:

- `mv /usr/bin/klog /usr/bin/klog.orig`



- `ln -s /usr/bin/klog.krb5 /usr/bin/klog`
4. Copiare il keytab da cresco-inst1:  
`cp /afs/enea.it/system/arc/keytab/services/host/cresco4x002.portici.enea.it /etc/krb5.keyta`
  5. Link simbolici alle shell? Discuterne con Guido
  6. Creare un link simbolico per pagsh:  
`ln -s /usr/bin/pagsh /usr/afsws/bin/pagsh`

### 4.3 Rimossi tra CentOS 6 e 7 e le cui alternative non presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 non è presente nei sistemi di CRESCO 6, e che si dovrebbe quindi provvedere ad installare:

Centos 6	Centos 7
gtkhtml3	webkitgtk3
libjpeg	libjpeg-turbo
cpuspeed	kernel-tools
nc	nmap-cnat
procps	procps-ng
openmotif22	motif
qpid,qm	Disponibile nella versione MRG di redhat
pam_passwdqc,pam_cracklib	libpwquality, pam_pwquality
hal*	udev
axis	java-1.7.0-openjdk
classpath[x]?-jaf	java-1.7.0-openjdk
classpath[x]?-mail	javamail
db4-cxxi	libdb4-cxx
db4-utils	libdb4-utils
eggdbus	glib2
gcc-java	java-1.7.0-openjdk-devel
GConf2-gtk	GConf2
geronimo-specs	geronimo-parent-poms
geronimo-specs-compatible	geronimo-jms, geronimo-jta
hal-devel	systemd-devel
ibus-gtk	ibus-gtk2
jakarta-commons-net	apache-commons-net
junit4	junit
m17n-contrib-*	m17n-contrib
m17n-db-*	m17n-db,m17n-db-extras
seekwatcher	iowatcher
udisks	udisks2
unique	unique2,glib2
unix2dos	dos2unix

#### 4.4 Rimossi da Centos 6 e 7 e le cui alternative sono presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 è presente nei sistemi di CRESCO 6, e che quindi non è necessario installare:

Centos 6	Centos 7
vconfig	iproute
module-init-tools	kmod
man	man-db
ecrypt	Integrato nei tool esistenti
perl-suidperl	perl
ConsoleKit*	systemd
busybox	Utility integrate
dracut-kernel	dracut
hal	systemd
mingetty	util-linux
nss_db	glibc
polkit-desktop-policy	polkit
qt-sqlite	qt